Proyecto

Carlos Calderón (15219)
Julio Barahona (15800)
Diego Castañeda (15151)

Matemática Discreta 2

Universidad del Valle de Guatemala
Facultad de Ingeniería
Guatemala
2016

Parte 1: Comparación de A* contra otro algoritmo

En nuestro caso se utilizó el algoritmo de dijkstra como algoritmo de comparación con A*. Para la primera parte de la experimentación, se observaron situaciones bastantes interesantes. En primer lugar se utilizó, un tablero de 15x15. Esto con el fin de que los algoritmos pudieran desenvolverse de una forma rápida y eficiente. Así pues se obtuvieron los siguientes resultados:

Algoritmo	Iteraciones	Costo
A*	42	28
*D*ijkstra	225	28

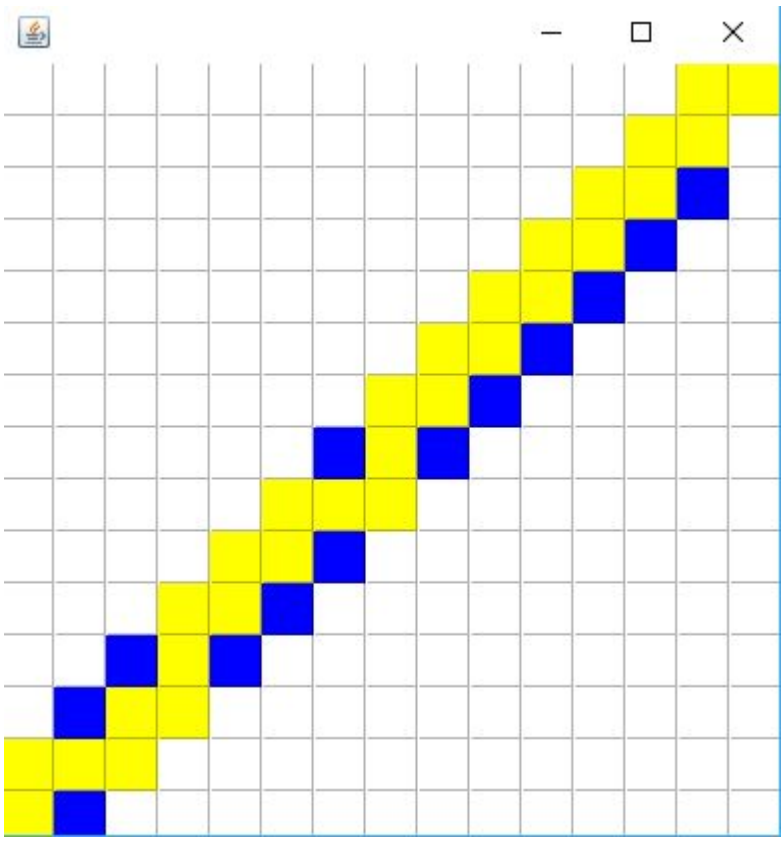

Imagen 1. Algoritmo de A en Grid de 15x 15*

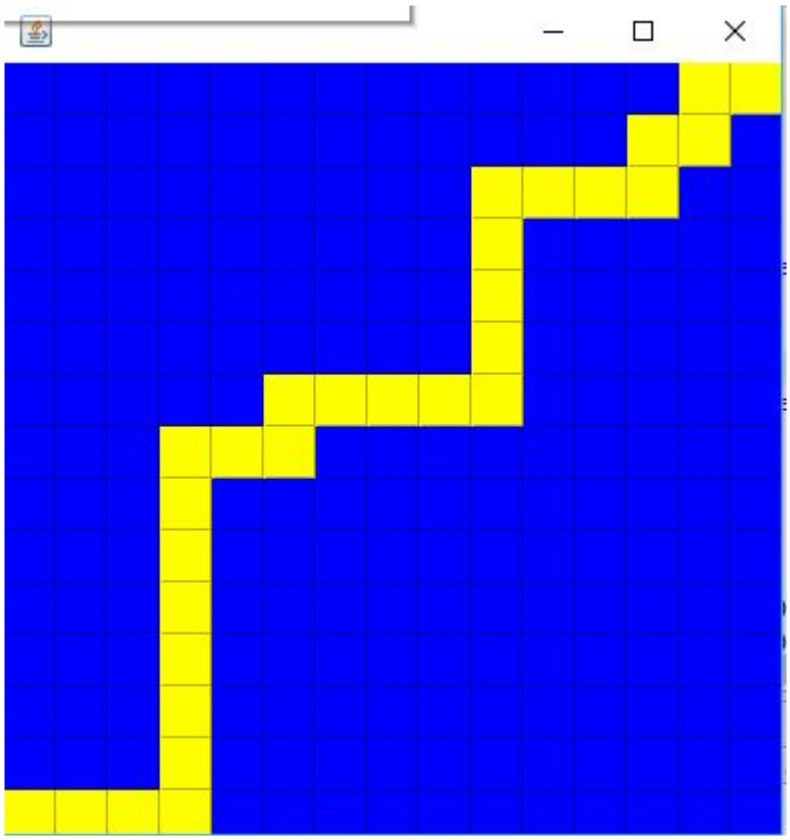

Imagen 2. Algoritmo de Dijkstra en Grid de 15x 15

Tal y como se esperaba, a pesar de dijkstra ser un caso especial de A*. A* fue más eficiente en cuanto a número de iteraciones. Pues su enfoque es buscar el camino más corto entre los nodos meta, y no entre cada nodo como dijkstra lo hace.

Para la segunda parte, se creó una función que juntaba a la función propuesta en la rúbrica junto con la función Manhattan, Además de ello, se creó una función cuyo propósito es ver en donde es más barato cambiar de nodo. Esta función propone al nodo del parámetro y su peso en comparación a sus vecinos y sus posibles pesos, con el objetivo de tomar el camino óptimo, considerando a futuro en donde sería más barato cambiar de nodos.

Algoritmo	Iteraciones	Costo
A*	79	174
*D*ijkstra	225	105

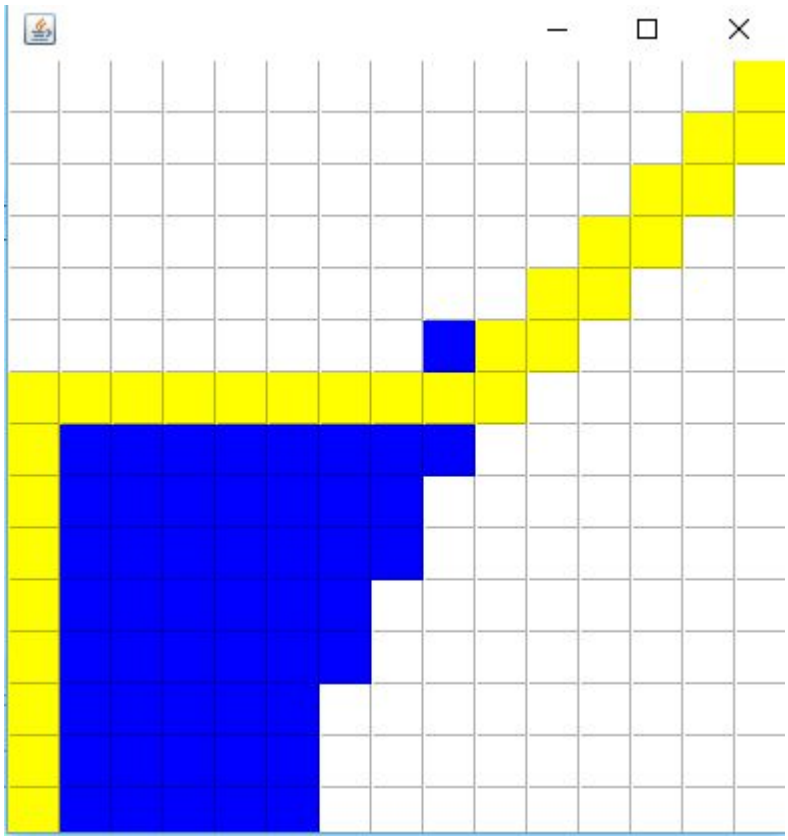

Imagen 3. Algoritmo de A en Grid de 15x 15 (con peso)*

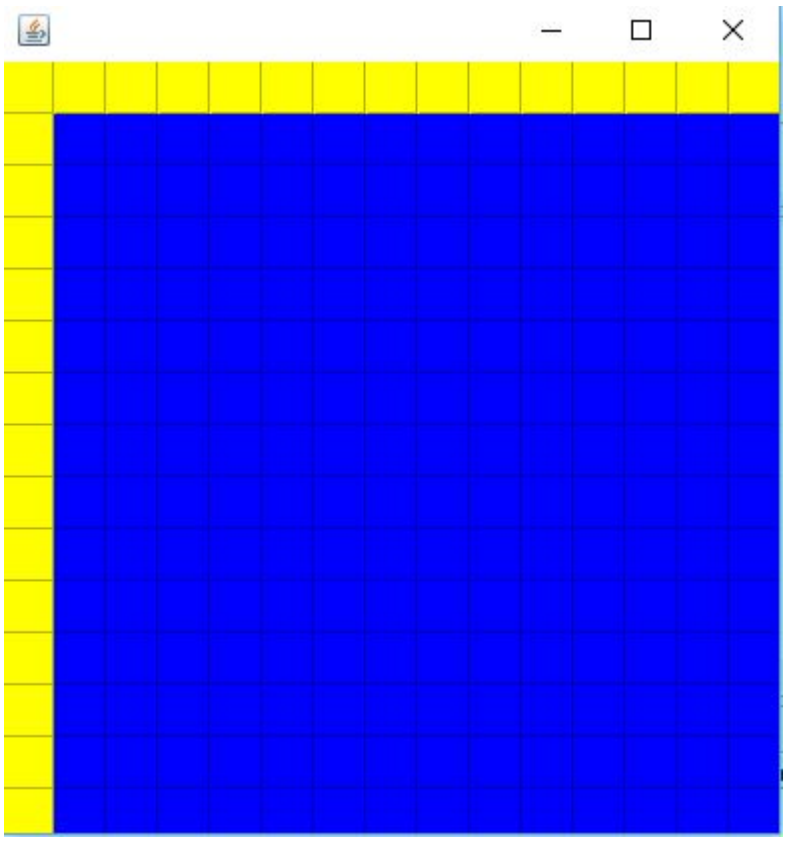

Imagen 4. Algoritmo de Dijkstra en Grid de 15x 15 (con peso)

Para la tercera parte, que constaba de una rejilla con aristas diagonales. Y con costo de $\sqrt{2}$ al recorrer dichas diagonales. En cuanto a costo, A* tuvo un pequeño aumento de 0.59 en cuanto a costo, haciendo a Dijkstra un poco más barato en cuanto al recorrido que tomó. Lo que hace, en este caso, a A* especial es que solo tuvo que iterarse 30 veces, mientras que Dijkstra recorrió todo el grafo para hallar el mejor camino. Se obtuvieron los siguientes resultados.

Algoritmo	Iteraciones	Costo
A*	30	22.14
dijkstra	225	21.55

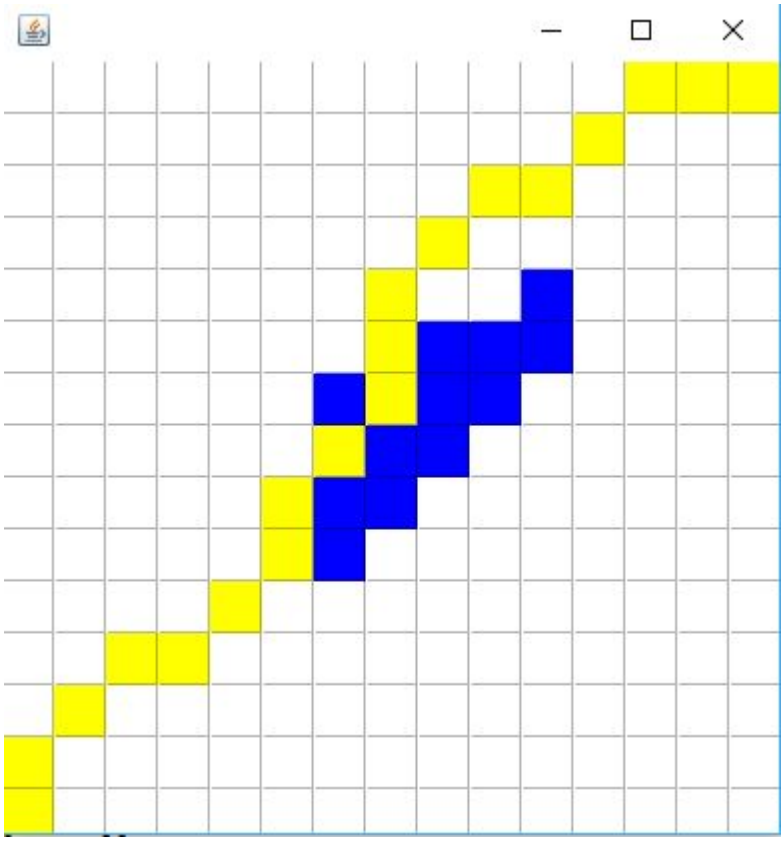

Imagen 5. Algoritmo de A con uso de Diagonales*

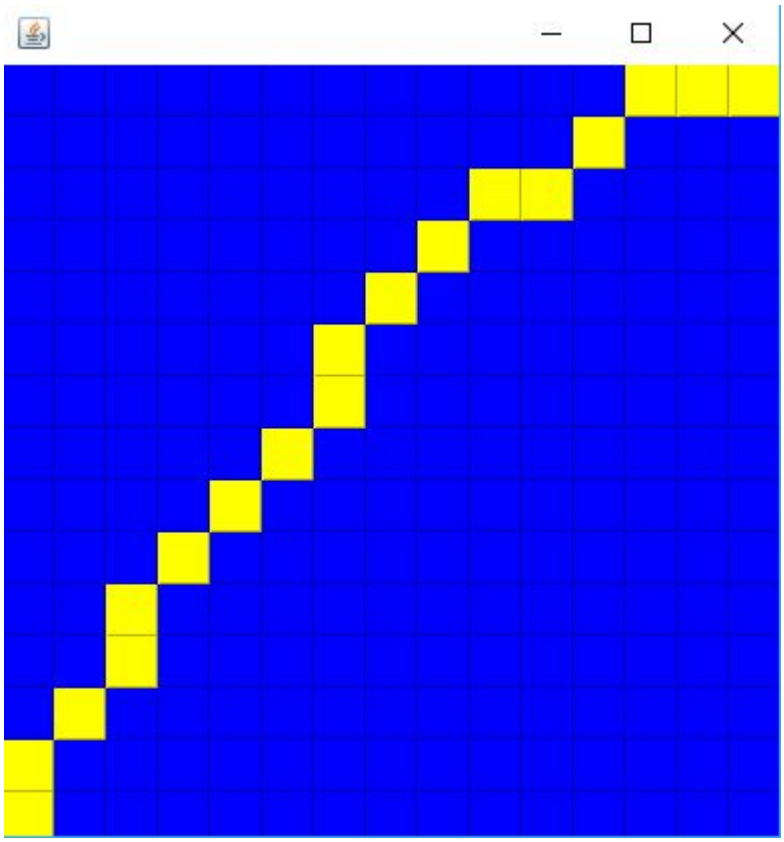

Imagen 6. Algoritmo de Dijkstra con uso de Diagonales de 15 x 15

Para la última parte de esta sección se compararon A* junto con Dijkstra. El resultado es similar a los anteriores, donde Dijkstra tuvo que evaluar a todos los nodos que no tuvieran el Obstáculo como true dentro de sus especificaciones. A*, a su diferencia, empieza a recorrer el camino usual de ir uno arriba y uno abajo hasta encontrarse con el obstáculo. Al momento de encontrarse con un tope, encuentra las estructuras más cercanas a las que puede ir para atravesar el obstáculo sin alejarse mucho del objetivo. Que era recorrer la rejilla con un obstáculo. Se obtuvieron los siguientes resultados:

Algoritmo	Iteraciones	Costo
A*	92	28
dijkstra	205	28

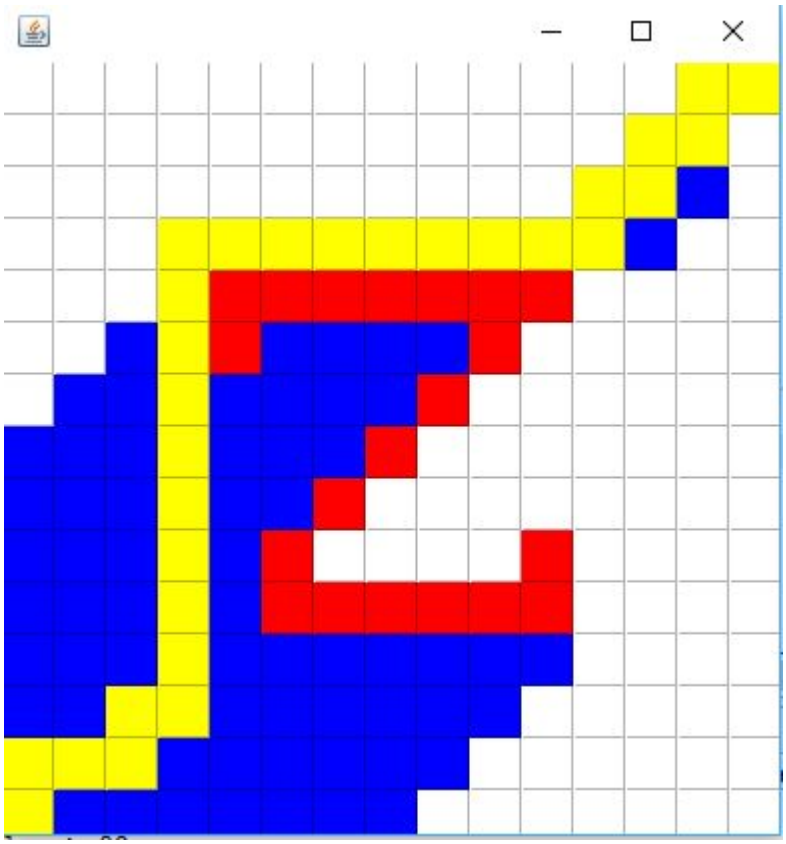

Imagen 7. Algoritmo de A con uso de Obstáculos*

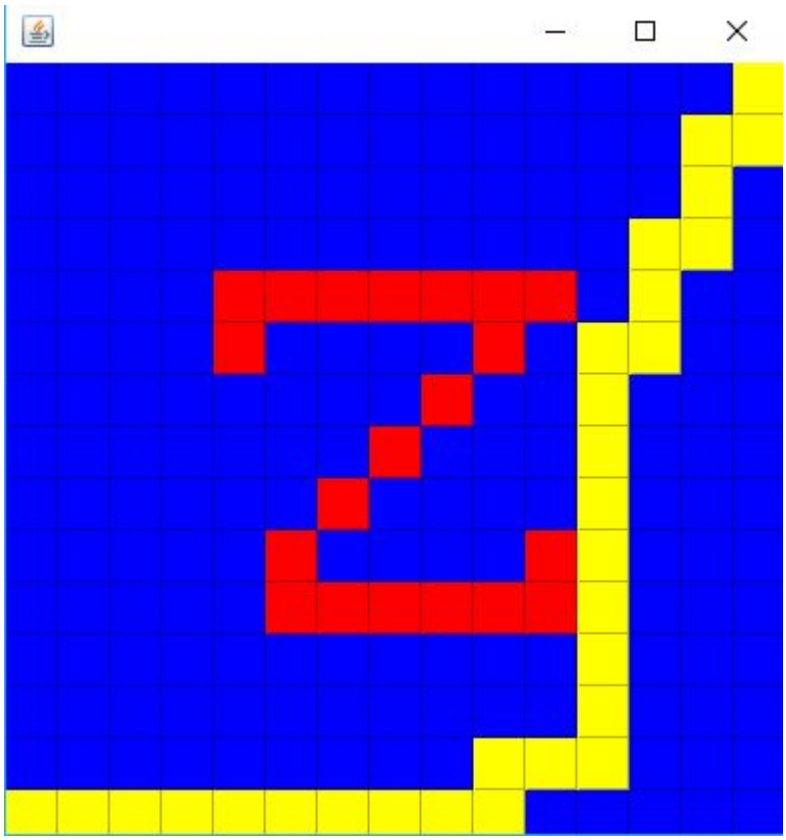

Imagen 8. Algoritmo de Dijkstra con uso de Obstáculos

Parte 2: Construcción de un laberinto y solución con A*

Referencias

Para esta parte de la experimentación se tomó una rejilla de 40x60. Esto simplemente se hizo así, ya que fue desarrollado en un lenguaje distinto al de la primera parte. Con este tamaño se vería un poco mejor la interfaz gráfica. Por lo que naturalmente, aquí el proceso fue un poco más cargado. Para generar el laberinto en base a la rejilla, se utilizó el algoritmo de prim. Cabe mencionar que para lograr que el programa en cada ejecución devolviera un laberinto distinto, se utilizó la versión aleatoria de Prim.

El algoritmo de Prim, según definición, es un algoritmo perteneciente a la teoría de grafos. Su función es la de encontrar un árbol mínimo recubridor en un grafo conexo, no dirigido y cuyas aristas están etiquetadas. El árbol recubridor mínimo es un subgrafo del grafo al que se le propone en donde se toman en cuenta todos los nodos del grafo sin crear ningún ciclo en el. Esto nos permitió crear, desde un nodo "raíz", las divisiones de todo el laberinto, las cuales se crean aleatoriamente ya que los pesos eran los mismos en todas las aristas.

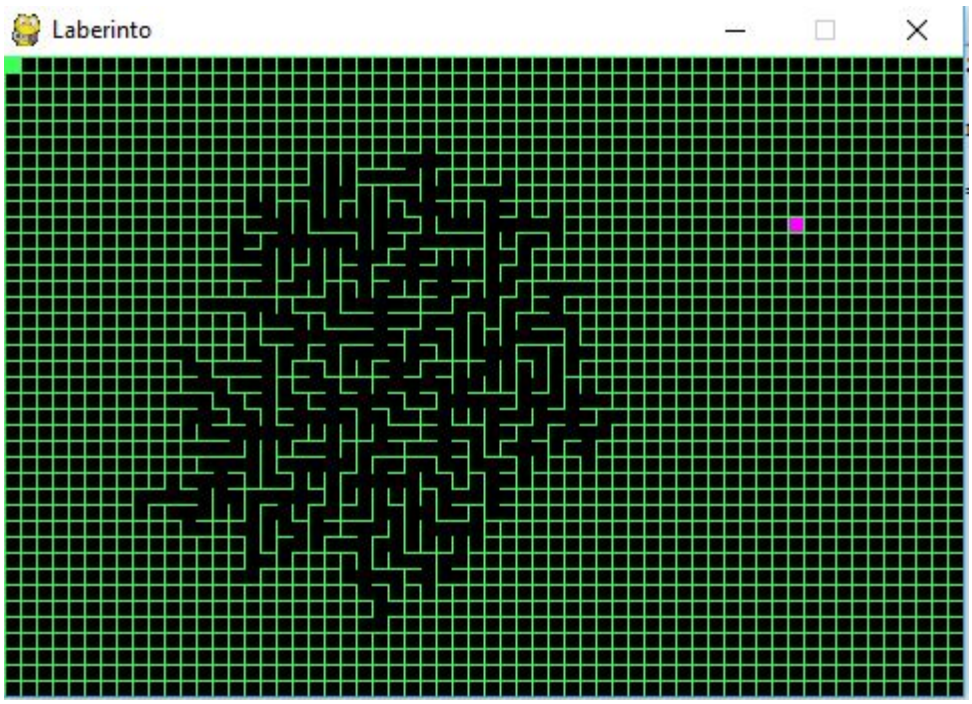

Imagen 9. Algoritmo de Prim generado laberinto

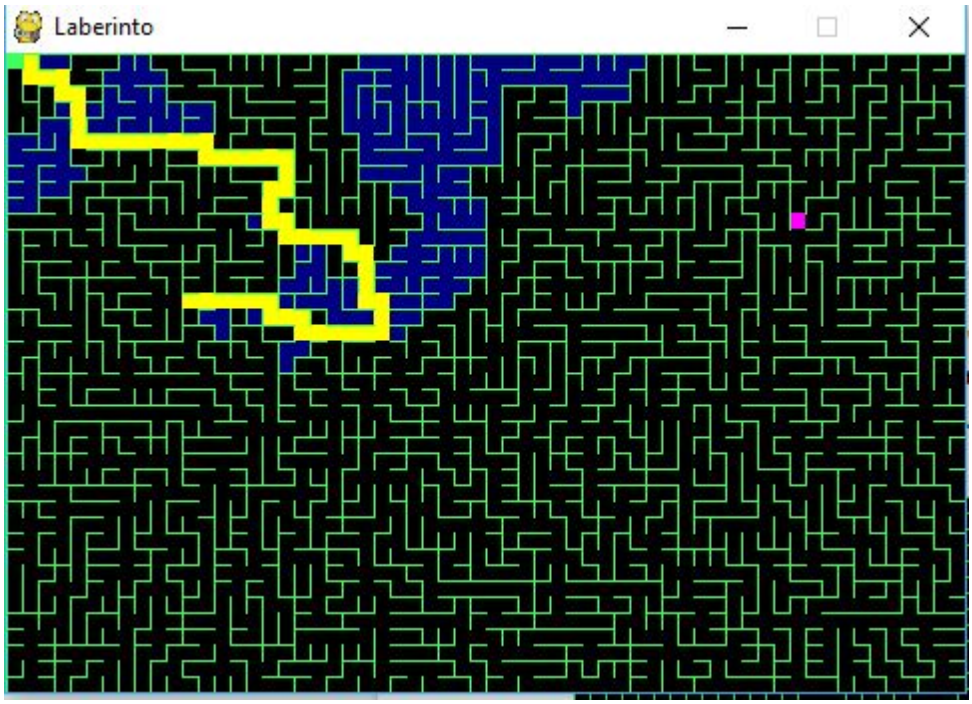

Imagen 10. DFS generando camino.

Posteriormente para conectar 2 puntos que el usuario ha escogido previamente. Como Prim, genera un árbol recubridor, entonces se utilizó el algoritmo “Depth-First Search”, que ejecuta búsquedas en en el mismo. Este algoritmo permite recorrer todos los nodos del grafo de una parte ordenada, más no uniforme.Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado. Utilizando este método, se puede observar que aunque sí hay una garantía de encontrar el nodo meta. Este algoritmo no siempre da el camino más óptimo. En muchas ciertas ocasiones llega a recorrer casi todo el laberinto, cuando un nodo está cercano.

Con esto se pudo comprender la repercusión que puede tener A* en la búsqueda de caminos. Pues implementando un tipo de algoritmo como este, aunque bajo ciertas condiciones puede tener un mejor rendimiento que A*. Este algoritmo no es muy inteligente.

Referencias

- Rajiv Eranki. 2002. Pathfinding using A* (A-Star). En : http://web.mit.edu/eranki/www/tutorials/search/. [Con acceso el 16 de noviembre de 2016].
- Depth-first search. 2016. En Wikipedia. Recuperado el 16 del 11 de 2016 de https://en.wikipedia.org/wiki/Depth-first_search.
- M.Oscar, & O. Maruja. 2008. Equinoccio. Grafos y Algoritmos.Recuperado el 16 del 11 de 2016.

- Maze generation algorithm. 2016. En Wikipedia. Recuperado el 16 del 11 de 2016 de https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_Prim.27s_algorithm.